
RABBITMQ 实战



<http://www.51mmr.net>

QQ 官方学习群: 651661013

1.0 RabbitMQ

- MQ为Message Queue，消息队列是应用程序和应用程序之间的通信方法。
- RabbitMQ是一个开源的，在AMQP基础上完整的，可复用的企业消息系统。
- 支持主流的操作系统，Linux、Windows、MacOX等。
- 多种开发语言支持，Java、Python、Ruby、.NET、PHP、C/C++、node.js等

开发语言：Erlang – 面向并发的编程语言。

AMQP:是消息队列的一个协议。

mysql 是 java 写的吗?不是 那么 java 能不能访问?可以,则通过(驱动)协议;那么要访问 RabbitMQ 是不是也可以通过驱动来访问

1.0.1 官网

http://www.rabbitmq.com/



1.0.2 其他 MQ



Apache RocketMQ

Apache RocketMQ™ is an open source distributed messaging and streaming data platform.

1.0.3 六种消息类型

http://www.rabbitmq.com/getstarted.html

RabbitMQ Tutorials

These tutorials cover the basics of creating messaging applications using RabbitMQ. You need to have the RabbitMQ server installed to go through the tutorials, please see the [installation guide](#). Code examples of these tutorials [are open source](#), as is [RabbitMQ website](#).

1 "Hello World!"

The simplest thing that does something

Python
Java
Ruby
PHP
C#
JavaScript
Go
Elixir
Objective-C
Swift
Spring AMQP

2 Work queues

Distributing tasks among workers the [competing consumers pattern](#)

Python
Java
Ruby
PHP
C#
JavaScript
Go
Elixir
Objective-C
Swift
Spring AMQP

3 Publish/Subscribe

Sending messages to many consumers at once

Python
Java
Ruby
PHP
C#
JavaScript
Go
Elixir
Objective-C
Swift
Spring AMQP

4 Routing

Receiving messages selectively

Python
Java
Ruby
PHP
C#
JavaScript
Go
Elixir
Objective-C
Swift
Spring AMQP

5 Topics

Receiving messages based on a pattern (topics)

Python
Java
Ruby
PHP
C#
JavaScript
Go
Elixir
Objective-C
Swift
Spring AMQP

6 RPC

[Request/reply pattern](#) example

Python
Java
Ruby
PHP
C#
JavaScript
Go
Elixir
Spring AMQP


1.0.4 安装文档

参考

名称	修改日期
1.otp_win64_20.2.exe	2018/1/22 14:43
2.rabbitmq-server-3.7.2.exe	2018/1/22 14:24
RabbitMQ-3.7.3安装手册.docx	2018/1/25 15:08
rabbitmq-server-3.7.2-1.el6.noarch.r...	2018/1/25 14:33
rabbitmq-server-3.7.2-1.el7.noarch.r...	2018/1/25 14:15

2.0 添加用户

2.1. 添加用户界面



3.7.2Erlang 20.2

OverviewConnectionsChannelsExchangesQueuesAdmin

Users

All users

Filter: ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/	•

?

Add a user

Username:

Password: (confirm)

Tags:

Set

Admin | Monitoring | Policymaker Management | Impersonator | None

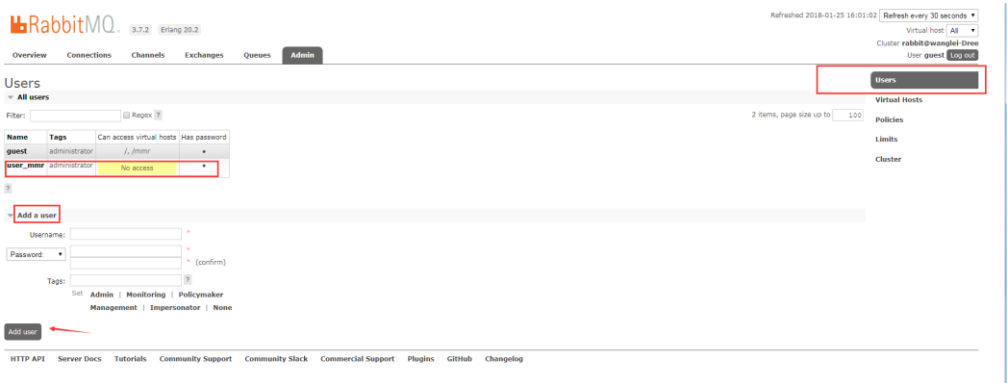
Add user

HTTP APIServer DocsTutorialsCommunity SupportCommunity SlackCommercial SupportPluginsGitHubChangelog

角色

2.2 添加管理员用户

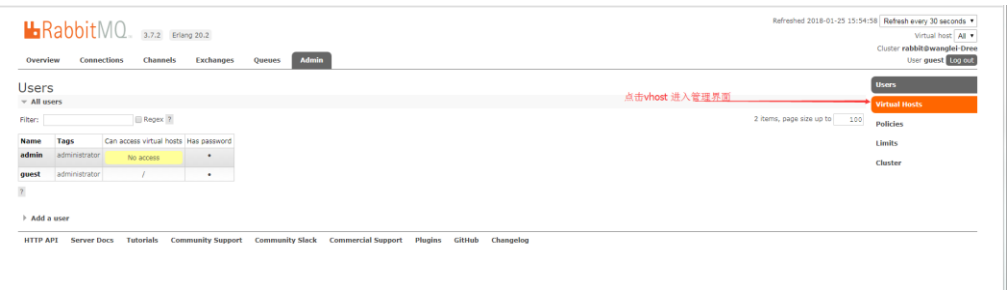
我们添加账号 user_mmr 密码 admin tags 选择 admin



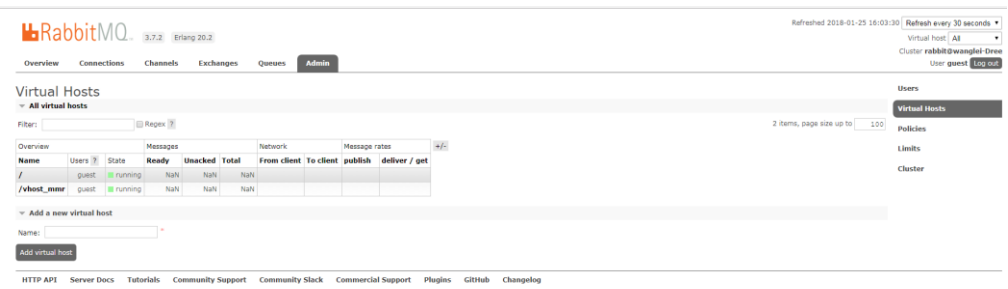
我们看到刚添加完成的用户 在 vhost 一栏是没有权限的, 所以呢我们这个时候的给他设置一个 vhost, 那么这个 vhost 就相当于一个数据库(可以理解为 mysql 里面的一个 db), 我们创建一个用户对其用户授权, 他就可以访问了

2.3 vhost 管理

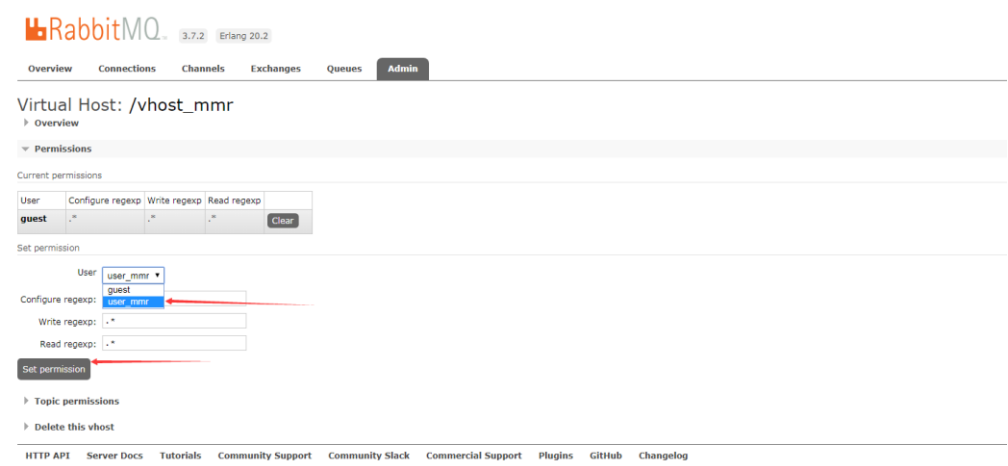
1. 点击右侧的菜单进入 vhost 的管理界面



2.点击 Add a new virtual host 添加一个 vhost,在 Rabbitmq 中我们添加 vhost 一般是以"/"开头,那么我们添加一个 /mmr 的 vhost;



3.当我们创建这个“vhost_mmr”的 vhost, 就可以对他进行用户授权,我们点击/vhost_mmr,进入其配置界面



在 permission 权限这一栏 我们选择刚刚创建的用户 user_mmr,选择完成后 Set Permission

RabbitMQ

3.7.2 Erlang 20.2

Overview

Connections

Channels

Exchanges

Queues

Admin

Users

All users

Filter: ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/, /vhost_mmr	•
user_mmr	administrator	/vhost_mmr	•

Add a user

Username:

Password: (confirm)

Tags:

Set **Admin** | **Monitoring** | **Policymaker**
Management | **Impersonator** | **None**

Add user

HTTP API

Server Docs

Tutorials

Community Support

Community Slack

Commercial Support

Plugins

GitHub

Changelog

我们退出 `guest` 用户,就可以使用刚刚创建的用户 `user_mmr` 进行登录了

RabbitMQ

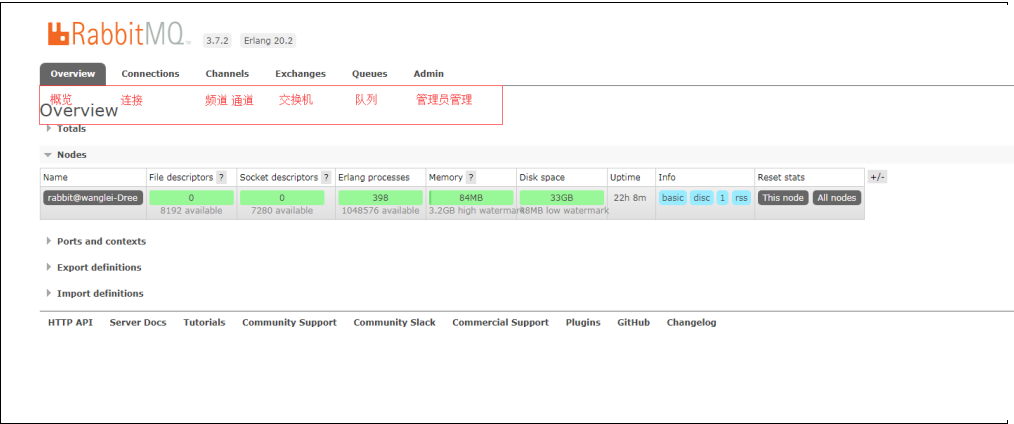
Username:

Password:

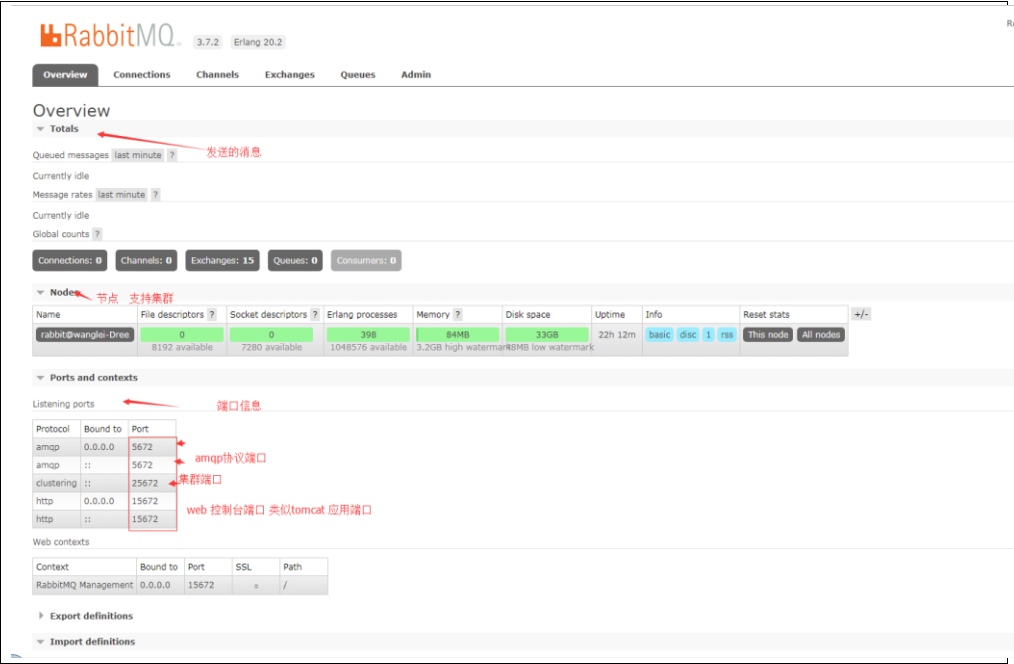
Login

这时候大功告成,创建成功!~

3.0 控制台功能介绍



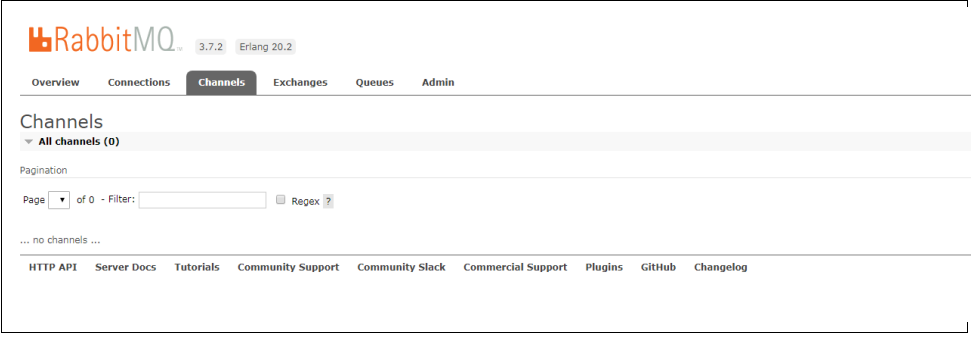
Overview 概览



Connections 连接



我们没有连接，这个就好像 jdbc 连接 mysql 一样 如果有程序连接这 ,这时候这里面就能显示哪些机器连接着



这些后面在讲具体的内容的时候 会提到是干什么用的

4.0 队列-java

4.1 项目准备

官方 demo 使用的是 4.0.2 版本

```
<dependencies>
  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>4.0.2</version>
  </dependency>

  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.10</version>
  </dependency>

  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
  </dependency>

  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
  </dependency>
</dependencies>
```

4.2 简单队列 hello world

4.2.1 模型图片



P: 消息的生产者

C: 消息的消费者

红色: 队列

生产者将消息发送到队列，消费者从队列中获取消息。

那么 we 根据以上的模型,咱们抽取出 3 个对象 **生产者**(用户发送消息) **队列(中间件)**:类似于容器(存储消息) 消费者(获取队列中的消息)

4.2.2 JAVA 操作 获取 MQ 连接

类似于我们在操作数据库的时候,的要获取到连接,然后才对数据进行操作

```
package com.mmr.rabbitmq.conn;
import java.io.IOException;
import java.util.concurrent.TimeoutException;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
public class ConnectionUtils {
    public static Connection getConnection() throws IOException, TimeoutException {
        //定义连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        //设置服务地址
        factory.setHost("127.0.0.1");
        //端口
        factory.setPort(5672); //amqp协议 端口 类似与mysql的3306
        //设置账号信息, 用户名、密码、vhost
        factory.setVirtualHost("/vhost_mmr");
        factory.setUsername("user_mmr");
        factory.setPassword("admin");
        // 通过工程获取连接
        Connection connection = factory.newConnection();
        return connection;
    }
}
```

4.2.3 生产者发送数据到消息队列

```
public class SendMQ {
    private static final String QUEUE_NAME="QUEUE_simple";
    /*
     P----->|QUEUE |
    */
    @Test
    public void sendMsg() throws Exception {
        /* 获取一个连接 */
        Connection connection = ConnectionUtils.getConnection();

        /*从连接中创建通道*/
        Channel channel = connection.createChannel();

        //创建队列（声明） 因为我们要往队列里面发送消息,这是后就得知往哪个队列中发送,就好比在哪个管子里面放水,
        boolean durable=false;
        boolean exclusive=false;
        boolean autoDelete=false;
        channel.queueDeclare(QUEUE_NAME, durable, exclusive, autoDelete, null);//如果这个队列不存在,其实这句话是不需要的

        String msg="Hello Simple QUEUE !";
        //第一个参数是exchangeName(默认情况下代理服务器端是存在一个""名字的exchange的,
        //因此如果不创建exchange的话我们可以直接将该参数设置成"",如果创建了exchange的话
        //我们需要将该参数设置成创建的exchange的名字),第二个参数是路由键
        channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
        System.out.println("-----send ms :"+msg);

        channel.close();
        connection.close();
    }
}
```

4.2.4 查看消息

RabbitMQ

3.7.2 Erlang 20.2

Overview

Connections

Channels

Exchanges

Queues

Admin

Queues

All queues (1)

Pagination

Page 1 of 1 - Filter: ☐ Regex

Overview

Messages

Message rates

+/-

Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/vhost_mmr	QUEUE_simple		idle	2	0	2	0.00/s		

Add a new queue

HTTP API

Server Docs

Tutorials

Community Support

Community Slack

Commercial Support

Plugins

GitHub

Changelog

RabbitMQ

3.7.2 Erlang 20.2

Overview

Connections

Channels

Exchanges

Queues

Admin

Operator policy

Consumer utilisation 0%

Message body bytes 63B

Process memory 13kB

Effective policy definition

Consumers

Bindings

Publish message

Get messages

Warning: getting messages from a queue is a destructive action.

Ack Mode: Ack message requeue false

Encoding: Auto string / base64

Messages: 1

Get Message(s)

Message 1

The server reported 2 messages remaining.

Exchange (AMQP default)

Routing Key QUEUE_simple

Redelivered 0

Properties

Payload 21 bytes Hello Simple QUEUE !

Encoding: string

Move messages

Delete

Purge

Runtime Metrics (Advanced)

4.2.5 消费者消费

```
package com.mmr.rabbitmq.simple;

import java.io.IOException;

import com.mmr.rabbitmq.conn.ConnectionUtils;
import com.rabbitmq.client.AMQP.BasicProperties;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.QueueingConsumer;

public class Consumer {
    private static final String QUEUE_NAME = "QUEUE_simple";

    public static void main(String[] args) throws Exception {

        /* 获取一个连接 */
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();

        //声明队列 如果能确定是哪一个队列 这边可以删掉,不去掉 这里会忽略创建
        //channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        DefaultConsumer consumer = new DefaultConsumer(channel) {
            //获取到达的消息
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties
properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received '" + message + "'");
            }
        };

        //监听队列
        channel.basicConsume(QUEUE_NAME, true, consumer);
    }

    @SuppressWarnings("deprecation")
    private static void oldGet(Channel channel) throws IOException, InterruptedException {
        //定义队列的消费者
        QueueingConsumer consumer = new QueueingConsumer(channel);
        // 监听队列
        channel.basicConsume(QUEUE_NAME, true, consumer);
        // 获取消息
        while (true) {
```

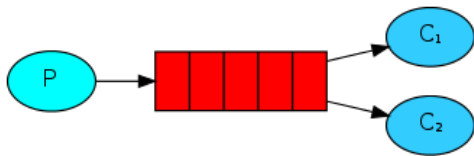
```
QueueingConsumer.Delivery delivery = consumer.nextDelivery();
String message = new String(delivery.getBody());
System.out.println(" [x] Received '" + message + "'");
    }
}
}
```

4.2.6 简单队列的不足

耦合性高 生产消费一一对应(如果有多个消费者想都消费这个消息,就不行了) 队列名称变更时需要同时更改

4.3 work queues 工作队列

4.3.0 模型图



为什么会出现 work queues?

前提:使用 simple 队列的时候

我们应用程序在是使用消息系统的时候,一般生产者 P 生产消息是毫不费力的(发送消息即可),而消费者接收完消息后的需要处理,会耗费一定的时间,这时候,就有可能导致很多消息堆积在队列里面,一个消费者有可能不够用

那么怎么让消费者同事处理多个消息呢?

在同一个队列上创建多个消费者,让他们相互竞争,这样消费者就可以同时处理多条消息了

使用任务队列的优点之一就是可以轻易的并行工作。如果我们积压了好多工作 ,我们可以通过增加工作者(消费者)

来解决这一问题 ,使得系统的伸缩性更加容易。

4.3.1 Round-robin（轮询分发）

4.3.2 生产者发送消息

```
public class Send {

    private final static String QUEUE_NAME = "test_queue_work";

    public static void main(String[] argv) throws Exception {
        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();

        // 声明队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        for (int i = 0; i < 50; i++) {
            // 消息内容
            String message = "." + i;
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
            System.out.println(" [x] Sent '" + message + "'");

            Thread.sleep(i * 10);
        }

        channel.close();
        connection.close();
    }
}
```

4.3.3 消费者 1

```
package com.mmr.rabbitmq.work;

@SuppressWarnings("deprecation")
public class Recv1 {

    private final static String QUEUE_NAME = "test_queue_work";

    public static void main(String[] args) throws Exception {
        // 获取到连接以及mq通道
```



```

Connection connection = ConnectionUtils.getConnection();
final Channel channel = connection.createChannel();

// 声明队列，主要为了防止消息接收者先运行此程序，队列还不存在时创建队列。
channel.queueDeclare(QueueName, false, false, false, null);

//定义一个消息的消费者
final Consumer consumer = new DefaultConsumer(channel) {
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties
properties, byte[] body) throws IOException {
        String message = new String(body, "UTF-8");
        System.out.println(" [1] Received '" + message + "'");
        try {
            doWork(message);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println(" [x] Done");
        }
    }
};

boolean autoAck = true; //消息的确认模式自动应答
channel.basicConsume(QueueName, autoAck, consumer);
}

private static void doWork(String task) throws InterruptedException {
    Thread.sleep(1000);
}

@SuppressWarnings("unused")
public static void oldAPI() throws Exception, TimeoutException {
    // 获取到连接以及mq通道
    Connection connection = ConnectionUtils.getConnection();
    Channel channel = connection.createChannel();

    // 声明队列
    channel.queueDeclare(QueueName, false, false, false, null);
    // 定义队列的消费者
    QueueingConsumer consumer = new QueueingConsumer(channel);
    // 监听队列，手动返回完成状态false 自动true 自动应答 不需要手动确认
    channel.basicConsume(QueueName, true, consumer);
    // 获取消息
    while (true) {
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();
        String message = new String(delivery.getBody());
        System.out.println(" [x] Received '" + message + "'");
    }
}

```

```
}  
  
}
```

4.3.4 消费者 2

```
public class Recv2 {  
  
    private final static String    QUEUE_NAME    = "test_queue_work";  
  
    public static void main(String[] args) throws Exception {  
        // 获取到连接以及mq通道  
        Connection connection = ConnectionUtils.getConnection();  
        final Channel channel = connection.createChannel();  
  
        // 声明队列  
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
  
        //定义一个消息的消费者  
        final Consumer consumer = new DefaultConsumer(channel) {  
            @Override  
            public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties  
properties, byte[] body) throws IOException {  
                String message = new String(body, "UTF-8");  
                System.out.println(" [2] Received '" + message + "'");  
                try {  
                    Thread.sleep(2000);  
                } catch (Exception e) {  
                    e.printStackTrace();  
                } finally {  
                    System.out.println(" [x] Done");  
                }  
            }  
        };  
  
        boolean autoAck = true; //  
        channel.basicConsume(QUEUE_NAME, autoAck, consumer);  
    }  
}
```

4.3.5 测试

备注:消费者 1 我们处理时间是 1s;而消费者 2 中处理时间是 2s;
但是我们看到的现象并不是 1 处理的多 消费者 2 处理的少,
[1] Received '.0'

```
[x] Done
[1] Received '.2'
[x] Done
[1] Received '.4'
[x] Done
[1] Received '.6'
.....
```

消费者 1 中将偶数部分处理掉了

```
[2] Received '.1'
[x] Done
[2] Received '.3'
[x] Done
[2] Received '.5'
[x] Done
.....
```

消费者 2 中将基数部分处理掉了

我想要的是 1 处理的多,而 2 处理的少

测试结果:

1.消费者 1 和消费者 2 获取到的消息内容是不同的,同一个消息只能被一个消费者获取

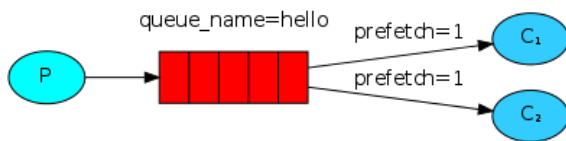
2.消费者 1 和消费者 2 货到的消息数量是一样的 一个奇数一个偶数

按道理消费者 1 获取的比消费者 2 要多

这种方式叫做**轮询分发** 结果就是不管谁忙或清闲,都不会给谁多一个任务或少一个任务,任务总是你一个我一个

的分

4.3.2 Fair dispatch（公平分发）



虽然上面的分配法方式也还行,但是有个问题就是:比如:现在有 2 个消费者,所有的偶数的消息都是繁忙的,而奇数则是轻松的。按照轮询的方式,偶数的任务交给了第一个消费者,所以一直在忙个不停。奇数的任务交给另一个消费者,则立即完成任务,然后闲得不行。

而 RabbitMQ 则是不了解这些的。他是不知道你消费者的消费能力的,这是因为当消息进入队列,RabbitMQ 就会分派消息。而 rabbitmq 只是盲目的将消息轮询的发给消费者。你一个我一个的这样发送。

为了解决这个问题，我们使用 `basicQos(prefetchCount = 1)`方法，来限制 RabbitMQ 只发不超过 1 条的消息给同一个消费者。当消息处理完毕后，有了反馈 `ack`，才会进行第二次发送。(也就是说需要手动反馈给 Rabbitmq)

还有一点需要注意，使用公平分发，必须关闭自动应答，**改为手动应答。**

4.3.2.1 生产者

```
public class Send {

    private final static String QUEUE_NAME = "test_queue_work";

    public static void main(String[] argv) throws Exception {
        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        // 创建一个频道
        Channel channel = connection.createChannel();
        // 指定一个队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        int prefetchCount = 1;

        //每个消费者发送确认信号之前，消息队列不发送下一个消息过来，一次只处理一个消息
        //限制发给同一个消费者不得超过1条消息
        channel.basicQos(prefetchCount);

        // 发送的消息
        for (int i = 0; i < 50; i++) {
            String message = "." + i;
            // 往队列中发出一条消息
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
            System.out.println(" [x] Sent '" + message + "'");
            Thread.sleep(i * 10);
        }
        // 关闭频道和连接
        channel.close();
        connection.close();
    }
}
```

4.3.2.2 消费者 1

```
public class Recv1 {
```

```

private final static String QUEUE_NAME = "test_queue_work";

public static void main(String[] args) throws Exception {
    // 获取到连接以及mq通道
    Connection connection = ConnectionUtils.getConnection();
    final Channel channel = connection.createChannel();

    // 声明队列，主要为了防止消息接收者先运行此程序，队列还不存在时创建队列。
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);

    channel.basicQos(1); // 保证一次只分发一个

    // 定义一个消息的消费者
    final Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties
properties, byte[] body) throws IOException {
            String message = new String(body, "UTF-8");
            System.out.println(" [1] Received '" + message + "'");
            try {
                doWork(message);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                System.out.println(" [x] Done");
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        }
    };

    boolean autoAck = false; // 手动确认消息
    channel.basicConsume(QUEUE_NAME, autoAck, consumer);
}

private static void doWork(String task) throws InterruptedException {
    Thread.sleep(1000);
}
}

```

4.3.2.3 消费者 2

```

@SuppressWarnings("deprecation")
public class Recv2 {

    private final static String QUEUE_NAME = "test_queue_work";

```

```

public static void main(String[] args) throws Exception {
    // 获取到连接以及mq通道
    Connection connection = ConnectionUtils.getConnection();
    final Channel channel = connection.createChannel();

    // 声明队列
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    channel.basicQos(1); // 保证一次只分发一个
    // 定义一个消息的消费者
    final Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties
properties, byte[] body) throws IOException {
            String message = new String(body, "UTF-8");
            System.out.println(" [2] Received '" + message + "'");
            try {
                doWork(message);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                System.out.println(" [x] Done");
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        }
    };

    boolean autoAck = false; // 关闭自动 确认
    channel.basicConsume(QUEUE_NAME, autoAck, consumer);
}

private static void doWork(String task) throws InterruptedException {
    Thread.sleep(2000);
}

public static void oldAPI() throws Exception, TimeoutException {
    // 获取到连接以及mq通道
    Connection connection = ConnectionUtils.getConnection();
    Channel channel = connection.createChannel();

    // 声明队列
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    // 同一时刻服务器只会发一条消息给消费者
    channel.basicQos(1);
    // 定义队列的消费者
    QueueingConsumer consumer = new QueueingConsumer(channel);
    // 监听队列，手动返回完成状态
    channel.basicConsume(QUEUE_NAME, false, consumer);
}

```

批注 [w1]:

批注 [w2]: 手动确认

批注 [w3]: 老 api 现在已经过期

批注 [w4]: False

```
// 获取消息
while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    String message = new String(delivery.getBody());
    System.out.println(" [x] Received '" + message + "'");
    // 休眠1秒
    Thread.sleep(1000);
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
}
}
```

批注 [w5]: 手动确认

这时候现象就是消费者 1 速度大于消费者 2

4.5 消息应答与消息持久化

4.5.1 Message acknowledgment（消息应答）

```
1. boolean autoAck = false;
2. channel.basicConsume(QUEUE_NAME, autoAck, consumer);
```

- `boolean autoAck = true;` (自动确认模式) 一旦 RabbitMQ 将消息分发给了消费者，就会从内存中删除。

在这种情况下，如果杀死正在执行任务的消费者，会丢失正在处理的消息，也会丢失已经分发给这个消费者但尚未处理的消息。

- `boolean autoAck = false;` (手动确认模式) 我们不想丢失任何任务，如果有一个消费者挂掉了，那么我们应该将分发给它的任务交付给另一个消费者去处理。为了确保消息不会丢失，RabbitMQ 支持消息应答。消费者发送一个消息应答，告诉 RabbitMQ 这个消息已经接收并且处理完毕了。RabbitMQ 可以删除它了。

- 消息应答是默认打开的。也就是 `boolean autoAck = false;`

4.5.2 Message durability（消息持久化）

我们已经了解了如何确保即使消费者死亡，任务也不会丢失。但是如果 RabbitMQ 服务器停止，我们的任务仍将失去！当 RabbitMQ 退出或者崩溃，将会丢失队列和消息。除非你不要队列和消息。两件事儿必须保证消息不被丢失：我们必须把“队列”和“消息”设为持久化。

```
1. boolean durable = true;
2. channel.queueDeclare("test_queue_work", durable, false, false, null);
```

那么我们直接将程序里面的 `false` 改成 `true` 就行了?? 不可以

会报异常 `channel error; protocol method: #method<channel.close>(reply-code=406, reply-text=PRECONDITION_FAILED - inequivalent arg 'durable' for queue 'test_queue_work')`

尽管这行代码是正确的，他不会运行成功。因为我们已经定义了一个名叫 `test_queue_work` 的未持久化的队列。RabbitMQ 不允许使用不同的参数设定重新定义已经存在的队列，并且会返回一个错误。

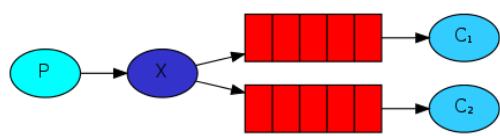
一个快速的解决方案——就是声明一个不同名字的队列，比如 `task_queue`。或者我们登录控制台将队列删除就可以了

批注 [w6]:

4.6 订阅模式 Publish/Subscribe

4.6.1 模型图

我们之前学习的都是一个消息只能被一个消费者消费,那么如果我想发一个消息 能被多个消费者消费,这时候怎么办? 这时候我们就得用到了消息中的发布订阅模型



在前面的教程中，我们创建了一个工作队列，都是一个任务只交给一个消费者。
这次我们做 将消息发送给多个消费者。这种模式叫做 “发布/订阅”。

举例：
类似微信订阅号 发布文章消息 就可以**广播给所有的接收者。(订阅者)**

那么咱们来看一下图,我们学过前两种有一些不一样,work 模式 是不是同一个队列 多个消费者,而 ps 这种模式呢,是一个队列对应一个消费者,pb 模式还多了一个 X(交换机 转发器),这时候我们要获取消息 就需要队列绑定到交换机上,交换机把消息发送到队列，消费者才能获取队列的消息

解读：

- 1、1 个生产者，多个消费者
- 2、每一个消费者都有自己的一个队列
- 3、生产者没有将消息直接发送到队列，而是发送到了交换机(转发器)
- 4、每个队列都要绑定到交换机
- 5、生产者发送的消息，经过交换机，到达队列，实现，一个消息被多个消费者获取的目的

注册完 发短信 发邮件
那么咱们先写一段代码爽一下

4.6.2 生产者



后台注册 → 邮件 → 短信

```
public class Send {  
  
    private final static String EXCHANGE_NAME = "test_exchange_fanout";  
  
    public static void main(String[] argv) throws Exception {  
        // 获取到连接以及mq通道  
        Connection connection = ConnectionUtils.getConnection();  
        Channel channel = connection.createChannel();  
        // 声明exchange 交换机 转发器
```

```
channel.exchangeDeclare(EXCHANGE_NAME, "fanout"); //fanout 分裂
// 消息内容
String message = "Hello PB";
channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes());
System.out.println(" [x] Sent '" + message + "'");

channel.close();
connection.close();
}
}
```

那么先看一下控制台 是不是有这个交换机

/vhost_mmr	amq.topic	topic			
/vhost_mmr	test_exchange_fanout	fanout		0.00/s	

但是这个发送的消息到哪了呢? 消息丢失了!!!因为交换机没有存储消息的能力,在 rabbitmq 中只有队列存储消息的能力,因为这时还没有队列,所以就会丢失;

小结:消息发送到了一个没有绑定队列的交换机时,消息就会丢失!

那么我们来写消费者

4.6.3 消费者 1

邮件发送系统

```
public class Recv {

    private final static String QUEUE_NAME = "test_queue_fanout_email";
    private final static String EXCHANGE_NAME = "test_exchange_fanout";

    public static void main(String[] argv) throws Exception {

        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        final Channel channel = connection.createChannel();

        // 声明队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        // 绑定队列到交换机
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "");
        //-----下面逻辑和work模式一样-----

        // 同一时刻服务器只会发一条消息给消费者
        channel.basicQos(1);
    }
}
```

```
// 定义一个消费者
Consumer consumer = new DefaultConsumer(channel) {
    // 消息到达 触发这个方法
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
        BasicProperties properties, byte[] body) throws IOException {

        String msg = new String(body, "utf-8");
        System.out.println("[1] Recv msg:" + msg);

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("[1] done ");
            // 手动回执
            channel.basicAck(envelope.getDeliveryTag(), false);
        }
    }
};

boolean autoAck = false;
channel.basicConsume(QUEUE_NAME, autoAck, consumer);
}
```

4.6.4 消费者 2

类似短信发送系统

```
public class Recv2 {

    private final static String QUEUE_NAME = "test_queue_fanout_2";

    private final static String EXCHANGE_NAME = "test_exchange_fanout";

    public static void main(String[] argv) throws Exception {

        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        final Channel channel = connection.createChannel();

        // 声明队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        // 绑定队列到交换机
```

```

channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "");

// 同一时刻服务器只会发一条消息给消费者
// 定义一个消费者
Consumer consumer = new DefaultConsumer(channel) {
    // 消息到达 触发这个方法
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
        BasicProperties properties, byte[] body) throws IOException {

        String msg = new String(body, "utf-8");
        System.out.println("[2] Recv msg:" + msg);

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("[2] done ");
            // 手动回执
            channel.basicAck(envelope.getDeliveryTag(), false);
        }
    }
};

boolean autoAck = false;
channel.basicConsume(QUEUE_NAME, autoAck, consumer);
}

```

测试

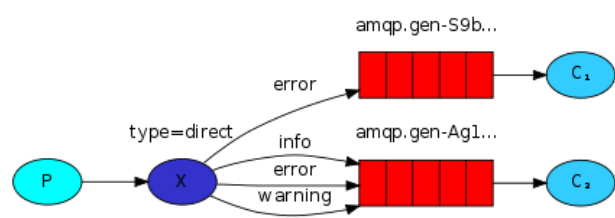
一个消息 可以被多个消费者获取

The screenshot shows the RabbitMQ Admin interface. The 'Exchanges' tab is selected. Under the 'Bindings' section, a red box highlights the bindings for 'test_exchange'. The bindings are as follows:

To	Routing key	Arguments	
test_queue_fanout_1			Unbind
test_queue_fanout_2			Unbind

4.7 路由模式

模型



生产者

```
public class Send {

    private final static String EXCHANGE_NAME = "test_exchange_direct";

    public static void main(String[] argv) throws Exception {
        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();

        // 声明exchange
        channel.exchangeDeclare(EXCHANGE_NAME, "direct");

        // 消息内容
        String message = "id=1001的商品删除了";
        channel.basicPublish(EXCHANGE_NAME, "delete", null, message.getBytes());
        System.out.println(" [x] Sent '" + message + "'");

        channel.close();
        connection.close();
    }
}
```

消费者 1

```
public class Recv {

    private final static String QUEUE_NAME = "test_queue_direct_1";
```

```

private final static String EXCHANGE_NAME = "test_exchange_direct";

public static void main(String[] argv) throws Exception {

    // 获取到连接以及mq通道
    Connection connection = ConnectionUtils.getConnection();
    final Channel channel = connection.createChannel();

    // 声明队列
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);

    // 绑定队列到交换机
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "update");
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "delete");

    // 同一时刻服务器只会发一条消息给消费者
    channel.basicQos(1);

    Consumer consumer = new DefaultConsumer(channel) {
        // 消息到达 触发这个方法
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
            BasicProperties properties, byte[] body) throws IOException {

            String msg = new String(body, "utf-8");
            System.out.println("[2] Recv msg:" + msg);

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                System.out.println("[2] done ");
                // 手动回执
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        }
    };

    boolean autoAck = false;
    channel.basicConsume(QUEUE_NAME, autoAck, consumer);
}
}

```

消费者 2

```
public class Recv2 {
    private static final String QUEUE_NAME="test_work_queue";

    public static void main(String[] args) throws IOException, TimeoutException {

        //获取连接
        Connection connection = ConnectionUtils.getConnection();
        //获取channel
        final Channel channel = connection.createChannel();
        //声明队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        channel.basicQos(1); //保证一次只分发一个

        //定义一个消费者
        Consumer consumer=new DefaultConsumer(channel){
            //消息到达 触发这个方法
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                BasicProperties properties, byte[] body) throws IOException {

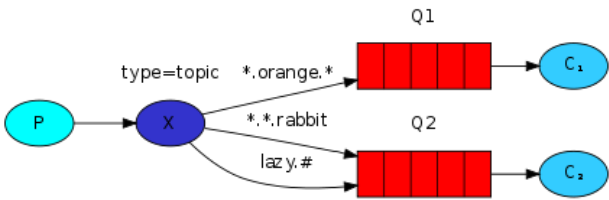
                String msg=new String(body,"utf-8");
                System.out.println("[2] Recv msg:"+msg);

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally{
                    System.out.println("[2] done ");
                    //手动回执
                    channel.basicAck(envelope.getDeliveryTag(), false);
                }
            }
        };

        boolean autoAck=false;
        channel.basicConsume(QUEUE_NAME,autoAck , consumer);
    }
}
```

4.7 Topic

模型



生产者

```
public class Send {

    private final static String EXCHANGE_NAME = "test_exchange_topic";

    public static void main(String[] argv) throws Exception {
        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();

        // 声明exchange
        channel.exchangeDeclare(EXCHANGE_NAME, "topic");

        // 消息内容
        String message = "id=1001";
        channel.basicPublish(EXCHANGE_NAME, "item.delete", null, message.getBytes());
        System.out.println(" [x] Sent '" + message + "'");

        channel.close();
        connection.close();
    }
}
```

消费者 1

```
public class Recv {

    private final static String QUEUE_NAME = "test_queue_topic_1";
```



```
private final static String EXCHANGE_NAME = "test_exchange_topic";

public static void main(String[] argv) throws Exception {

    // 获取到连接以及mq通道
    Connection connection = ConnectionUtils.getConnection();
    final Channel channel = connection.createChannel();

    // 声明队列
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);

    // 绑定队列到交换机
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "item.update");
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "item.delete");

    // 同一时刻服务器只会发一条消息给消费者
    channel.basicQos(1);

    // 定义队列的消费者
    Consumer consumer = new DefaultConsumer(channel) {
        // 消息到达 触发这个方法
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
            BasicProperties properties, byte[] body) throws IOException {

            String msg = new String(body, "utf-8");
            System.out.println("[2] Recv msg:" + msg);

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                System.out.println("[2] done ");
                // 手动回执
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        }
    };

    boolean autoAck = false;
    channel.basicConsume(QUEUE_NAME, autoAck, consumer);
}
```

消费者 2

```
public class Recv {

    private final static String QUEUE_NAME = "test_queue_topic_1";
    private final static String EXCHANGE_NAME = "test_exchange_topic";

    public static void main(String[] argv) throws Exception {

        // 获取到连接以及mq通道
        Connection connection = ConnectionUtils.getConnection();
        final Channel channel = connection.createChannel();

        // 声明队列
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        // 绑定队列到交换机
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "item.update");
        channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "item.delete");

        // 同一时刻服务器只会发一条消息给消费者
        channel.basicQos(1);

        // 定义队列的消费者
        Consumer consumer = new DefaultConsumer(channel) {
            // 消息到达 触发这个方法
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                      BasicProperties properties, byte[] body) throws IOException {

                String msg = new String(body, "utf-8");
                System.out.println("[2] Recv msg:" + msg);

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    System.out.println("[2] done ");
                    // 手动回执
                    channel.basicAck(envelope.getDeliveryTag(), false);
                }
            }
        };

        boolean autoAck = false;
        channel.basicConsume(QUEUE_NAME, autoAck, consumer);
    }
}
```

```
}  
}
```

4.8 Exchanges（转发器|交换机）

转发器一方面它接受生产者的消息，另一面向队列推送消息。

Nameless exchange（匿名转发）

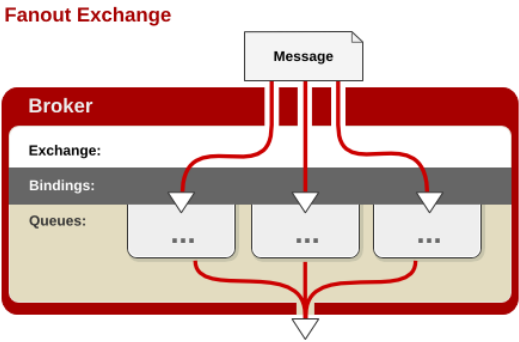
之前我们对转换器一无所知，却可以将消息发送到队列，那是可能是我们用了默认的转发器，转发器名为空字符串""。之前我们发布消息的代码是：

```
[java] view plain copy
```

```
1. channel.basicPublish("", "hello", null, message.getBytes());
```

Fanout Exchange

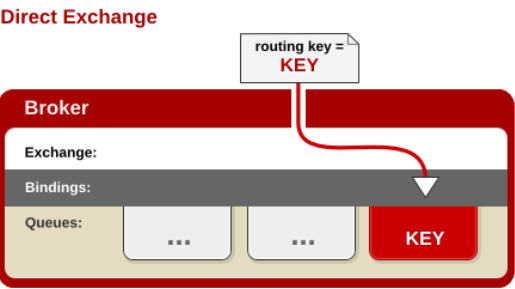
不处理路由键。你只需要将队列绑定到交换机上。发送消息到交换机都会被转发到与该交换机绑定的所有队列上。



Direct Exchange

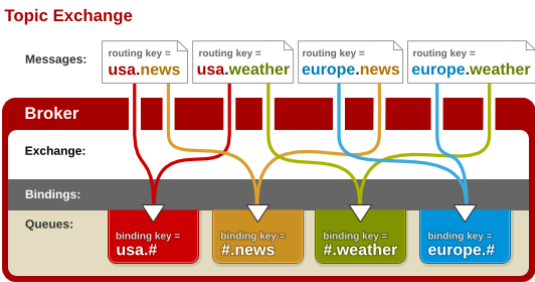
处理路由键。需要将一个队列绑定到交换机上，要求该消息与一个特定的路由键完全匹配。这是一个完整的匹配。如果一个队列绑定到该交换机上要求路由键 “dog”，则只有被标记为“dog”的消息才被转发，不会转发 dog.puppy，也不会转发

dog.guard，只会转发 dog。



Topic Exchange

将路由键和某模式进行匹配。
此时队列需要绑定要一个模式上。符号“#”匹配一个或多个词，符号“*”匹配一个词。因此“audit.#”能够匹配到“audit.irs.corporate”，但是“audit.*” 只会匹配到“audit.irs”。



4.9 RabbitMQ 之消息确认机制（事务+Confirm）

概述

在 Rabbitmq 中我们可以通过持久化来解决因为服务器异常而导致丢失的问题,

除此之外我们还会遇到一个问题:生产者将消息发送出去之后,消息到底有没有正确到达 Rabbit 服务器呢?如果不错得数处理,我们是不知道的,(即 Rabbit 服务器不会反馈任何消息给生产者),也就是默认的情况下是不知道消息有没有正确到达;

导致的问题:消息到达服务器之前丢失,那么持久化也不能解决此问题,因为消息根本就没有到达 Rabbit 服务器!

RabbitMQ 为我们提供了两种方式:

1. 通过 AMQP 事务机制实现，这也是 AMQP 协议层面提供的解决方案；
2. 通过将 channel 设置成 confirm 模式来实现；

事务机制

RabbitMQ 中与事务机制有关的方法有三个：txSelect(), txCommit()以及 txRollback(), txSelect 用于将当前 channel 设置成 transaction 模式，txCommit 用于提交事务，txRollback 用于回滚事务，在通过 txSelect 开启事务之后，我们便可以发布消息给 broker 代理服务器了，如果 txCommit 提交成功了，则消息一定到达了 broker 了，如果在 txCommit 执行之前 broker 异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过 txRollback 回滚事务了。

关键代码:

```
channel.txSelect();
channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
channel.txCommit();
```

生产者

```
public class SendMQ {
    private static final String QUEUE_NAME = "QUEUE_simple";

    @Test
    public void sendMsg() throws IOException, TimeoutException {
        /* 获取一个连接 */
        Connection connection = ConnectionUtils.getConnection();

        /* 从连接中创建通道 */
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        String msg = "Hello Simple QUEUE !";

        try {
            channel.txSelect();
            channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
            int result = 1 / 0;
            channel.txCommit();
        } catch (Exception e) {
            channel.txRollback();
            System.out.println("----msg rollback ");
        } finally{
```

```
        System.out.println("-----send msg over:" + msg);
    }
    channel.close();
    connection.close();
}
}
```

消费者

```
public class Consumer {
    private static final String QUEUE_NAME = "QUEUE_simple";

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtils.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        DefaultConsumer consumer = new DefaultConsumer(channel) {
            //获取到达的消息
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
BasicProperties properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received '" + message + "'");
            }
        };
        //监听队列
        channel.basicConsume(QUEUE_NAME, true, consumer);
    }
}
```

此种模式还是很耗时的,采用这种方式 降低了 Rabbitmq 的消息吞吐量

Confirm 模式

概述

上面我们介绍了 RabbitMQ 可能会遇到的一个问题,即生成者不知道消息是否真正到达 broker, 随后通过 AMQP 协议层面为我们提供了事务机制解决了这个问题, **但是采用事务机制实现会降低 RabbitMQ 的消息吞吐量**, 那么有没有更加高效的解决方式呢? 答案是采用 Confirm 模式。

producer 端 confirm 模式的实现原理

生产者将信道设置成 `confirm` 模式，一旦信道进入 `confirm` 模式，所有在该信道上发布的信息都会被指派一个唯一的 ID(从 1 开始)，一旦信息被投递到所有匹配的队列之后，`broker` 就会发送一个确认给生产者（包含信息的唯一 ID），这就使得生产者知道信息已经正确到达目的队列了，如果信息和队列是可持久化的，那么确认信息会将信息写入磁盘之后发出，`broker` 回传给生产者的确认信息中 `deliver-tag` 域包含了确认信息的序列号，此外 `broker` 也可以设置 `basic.ack` 的 `multiple` 域，表示到这个序列号之前的所有信息都已经得到了处理。

`confirm` 模式最大的好处在于他是异步的，一旦发布一条信息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条信息，当信息最终得到确认之后，生产者应用便可以通过回调方法来处理该确认信息，如果 `RabbitMQ` 因为自身内部错误导致信息丢失，就会发送一条 `nack` 消息，生产者应用程序同样可以在回调方法中处理该 `nack` 消息。

开启 confirm 模式的方法

已经在 transaction 事务模式的 channel 是不能再设置成 confirm 模式的，即这两种模式是不能共存的。

生产者通过调用 channel 的 confirmSelect 方法将 channel 设置为 confirm 模式

核心代码:

```
//生产者通过调用channel的confirmSelect方法将channel设置为confirm模式
channel.confirmSelect();
```

编程模式

1. 普通 confirm 模式：每发送一条消息后，调用 waitForConfirms()方法，等待服务器端 confirm。实际上是一种串行 confirm 了。
2. 批量 confirm 模式：每发送一批消息后，调用 waitForConfirms()方法，等待服务器端 confirm。
3. 异步 confirm 模式：提供一个回调方法，服务端 confirm 了一条或者多条消息后 Client 端会回调这个方法。

1.普通 confirm 模式

```
public class SendConfirm {

    private static final String  QUEUE_NAME  = "QUEUE_simple_confirm";

    @Test
    public void sendMsg() throws IOException, TimeoutException,
InterruptedException {
        /* 获取一个连接 */
        Connection connection = ConnectionUtils.getConnection();

        /* 从连接中创建通道 */
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        //生产者通过调用channel的confirmSelect方法将channel设置为confirm模式
        channel.confirmSelect();

        String msg = "Hello  QUEUE !";
```



```

channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());

if(!channel.waitForConfirms()){
    System.out.println("send message failed.");
}else{
    System.out.println(" send messgae ok ...");
}

channel.close();
connection.close();
}
}

```

2. 批量 confirm 模式

批量 confirm 模式稍微复杂一点，客户端程序需要定期（每隔多少秒）或者定量（达到多少条）或者两则结合起来 publish 消息，然后等待服务器端 confirm，相比普通 confirm 模式，批量极大提升 confirm 效率，但是问题在于一旦出现 confirm 返回 false 或者超时的情况时，客户端需要将这一批次的消息全部重发，这会带来明显的重复消息数量，并且，当消息经常丢失时，批量 confirm 性能应该是不升反降的。

```

public class SendbatchConfirm {

    private static final String QUEUE_NAME = "QUEUE_simple_confirm";

    @Test
    public void sendMsg() throws IOException, TimeoutException,
        InterruptedException {
        /* 获取一个连接 */
        Connection connection = ConnectionUtils.getConnection();

        /* 从连接中创建通道 */
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        //生产者通过调用channel的confirmSelect方法将channel设置为confirm模式
        channel.confirmSelect();

        String msg = "Hello QUEUE !";
        for (int i = 0; i < 10; i++) {
            channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
        }

        if(!channel.waitForConfirms()){
            System.out.println("send message failed.");
        }else{

```

```

        System.out.println(" send messgae ok ...");
    }

    channel.close();
    connection.close();
}
}

```

3.异步 confirm 模式

Channel 对象提供的 ConfirmListener()回调方法只包含 deliveryTag (当前 Chanel 发出的消息序号)，我们需要自己为每一个 Channel 维护一个 unconfirm 的消息序号集合，每 publish 一条数据，集合中元素加 1，每回调一次 handleAck 方法，unconfirm 集合删掉相应的一条 (multiple=false) 或多条 (multiple=true) 记录。从程序运行效率上看，这个 unconfirm 集合最好采用有序集合 SortedSet 存储结构。实际上，SDK 中的 waitForConfirms()方法也是通过 SortedSet 维护消息序号的。

```

public class SendAync {
    private static final String  QUEUE_NAME  = "QUEUE_simple_confirm_aync";

    public static void main(String[] args) throws IOException, TimeoutException {
        /* 获取一个连接 */
        Connection connection = ConnectionUtils.getConnection();

        /* 从连接中创建通道 */
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);

        //生产者通过调用channel的confirmSelect方法将channel设置为confirm模式
        channel.confirmSelect();

        final SortedSet<Long> confirmSet = Collections.synchronizedSortedSet(new
        TreeSet<Long>());

        channel.addConfirmListener(new ConfirmListener() {
            //每回调一次handleAck方法，unconfirm集合删掉相应的一条 (multiple=false)
            或多条 (multiple=true) 记录。
            @Override
            public void handleAck(long deliveryTag, boolean multiple) throws
            IOException {
                if (multiple) {
                    System.out.println("--multiple--");
                    confirmSet.headSet(deliveryTag + 1).clear();//用一个
                    SortedSet, 返回此有序集合中小于end的所有元素。
                } else {

```

```

        System.out.println("--multiple false--");
        confirmSet.remove(deliveryTag);
    }
}

@Override
public void handleNack(long deliveryTag, boolean multiple) throws
IOException {
    System.out.println("Nack, SeqNo: " + deliveryTag + ", multiple:
" + multiple);
    if (multiple) {
        confirmSet.headSet(deliveryTag + 1).clear();
    } else {
        confirmSet.remove(deliveryTag);
    }
}

});

String msg = "Hello  QUEUE !";
while (true) {
    long nextSeqNo = channel.getNextPublishSeqNo();
    channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
    confirmSet.add(nextSeqNo);
}

}
}

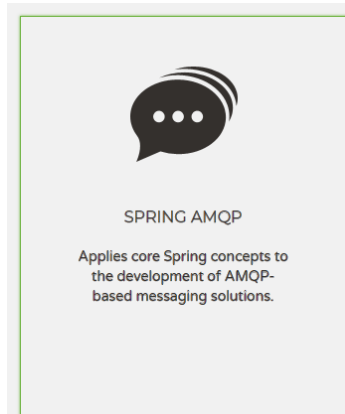
```

5.0 Spring rabbit

简介

为了更好更快速的去开发 Rabbitmq 应用, spring 封装 client ;目的是简化我们的使用.

<https://projects.spring.io/spring-amqp/>



Spring 对 amqp 的支持 目前只对 Rabbitmq 做了实现,所以我们可以使用 Rabbitmq 简化我们的开发;

The project consists of two parts; `spring-amqp` is the base abstraction, and `spring-rabbit` is the `RabbitMQ` implementation.

开发集成

maven 依赖

```
<dependencies>
  <dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
    <version>1.7.5.RELEASE</version>
  </dependency>
</dependencies>
```

生产者

```
public class SpringMain {
    public static void main(final String... args) throws Exception {
        AbstractApplicationContext ctx = new
        ClassPathXmlApplicationContext("classpath:context.xml");
        //RabbitMQ模板
        RabbitTemplate template = ctx.getBean(RabbitTemplate.class);
        //发送消息
    }
}
```

```
        template.convertAndSend("Hello, world!");  
        Thread.sleep(1000); // 休眠1秒  
        ctx.destroy(); //容器销毁  
    }  
}
```

消费者

```
public class MyConsumer {  
  
    //具体执行业务的方法  
    public void listen(String foo) {  
        System.out.println("消费者: " + foo);  
    }  
}
```

Context.xml 配置文件

```
<beans xmlns="http://www.springframework.org/schema/beans"  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
        xmlns:rabbit="http://www.springframework.org/schema/rabbit"  
        xsi:schemaLocation="http://www.springframework.org/schema/rabbit  
            http://www.springframework.org/schema/rabbit/spring-rabbit-1.4.xsd  
            http://www.springframework.org/schema/beans  
            http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">  
  
    <!--1.定义RabbitMQ的连接工厂 -->  
    <rabbit:connection-factory id="connectionFactory"  
        host="127.0.0.1" port="5672" username="user_mmr" password="admin"  
        virtual-host="/vhost_mmr" />  
  
    <!--2.定义Rabbit模板，指定连接工厂以及定义exchange -->  
    <rabbit:template id="amqpTemplate" connection-factory="connectionFactory"  
        exchange="fanoutExchange" />  
    如果不想将消息发送到交换机 可以将它设置成队列 将交换机删除  
    <!-- 定义Rabbit模板，指定连接工厂以及定义exchange -->  
    <rabbit:template id="amqpTemplate" connection-factory="connectionFactory" queue="xxxxq" />  
  
    <!-- MQ的管理，包括队列、交换器 声明等 -->  
    <rabbit:admin connection-factory="connectionFactory" />  
  
    <!-- 定义队列，自动声明 -->  
    <rabbit:queue name="myQueue" auto-declare="true" durable="true"/>
```

```
<!-- 定义交换器，自动声明 -->
<rabbit:fanout-exchange name="fanoutExchange" auto-declare="true">
  <rabbit:bindings>
    <rabbit:binding queue="myQueue"/>
  </rabbit:bindings>
</rabbit:fanout-exchange>

<!-- 队列监听 -->
<rabbit:listener-container connection-factory="connectionFactory">
  <rabbit:listener ref="foo" method="Listen" queue-names="myQueue" />
</rabbit:listener-container>

<!-- 消费者 -->
<bean id="foo" class="com.mmr.rabbitmq.spring.MyConsumer" />

</bean>
```